Experimental evaluation of two software countermeasures against fault attacks

Nicolas Moro, Karine Heydemann, Amine Dehbaoui, Bruno Robisson, Emmanuelle Encrenaz

Experimental evaluation of two software countermeasures against fault attacks

Nicolas Moro[1,2], Karine Heydemann[2], Amine Dehbaoui[3], Bruno Robisson[1], and Emmanuelle Encrenaz[2]

[1]CEA, CEA-Tech PACA, LSAS, 13541 Gardanne, France
nicolas.moro@cea.fr
[2]Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, 75005 Paris, France
karine.heydemann@lip6.fr
[3]SERMA Technologies, CESTI, 33615 Pessac, France
a.dehbaoui@serma.com

Abstract

Injection of transient faults can be used as a way to attack embedded systems. On embedded processors such as microcontrollers, several studies showed that such a transient fault injection with glitches or electromagnetic pulses could corrupt either the data loads from the memory or the assembly instructions executed by the circuit. Some countermeasure schemes which rely on temporal redundancy have been proposed to handle this issue. Among them, several schemes add this redundancy at assembly instruction level. In this paper, we perform a practical evaluation for two of those countermeasure schemes by using a pulsed electromagnetic fault injection process on a 32-bit microcontroller. We provide some necessary conditions for an efficient implementation of those countermeasure schemes in practice. We also evaluate their efficiency and highlight their limitations. To the best of our knowledge, no experimental evaluation of the security of such instruction-level countermeasure schemes has been published yet.

1 Introduction

Physical attacks were introduced in the late 1990s as a new way to break cryptosystems by exploiting weaknesses in their implementation. Among them, fault attacks were introduced by Boneh *et al.* in 1997 (Boneh, DeMillo, and Lipton 1997). Those attacks consist in applying a stress to the circuit in order to induce transient faults which could create an attack path (Barenghi, Breveglieri, Koren, and Naccache 2012). Such transient faults can be induced in a large set of embedded circuits by using many physical means which include circuit underpowering (Bhasin et al. 2009), clock glitches (Balasch, Gierlichs, and Verbauwhede 2011), voltage glitches (Zussa et al. 2013), changes in the temperature (Skorobogatov 2009) or laser shots (Trichina and Korkikyan 2010). More recently, two other fault injection techniques based on using electromagnetic waves have been proposed, either by using a harmonic injection signal (Poucheret et al. 2011) or by using electromagnetic glitches (Dehbaoui et al. 2012). These physical fault injection means enable to perform higher-level attack schemes such as Differential Fault Analysis (DFA) or safe-error attacks (Karaklajic, Schmidt, and Verbauwhede 2013).

This work was done while Amine Dehbaoui was with École Nationale Supérieure des Mines de Saint-Étienne (ENSM.SE), 13541 Gardanne, France.

This article has been presented at the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST 2014, Arlington, USA). A copy of the presentation can be found on http://www.nicolasmoro.net/research.

Those higher-level attack schemes all rely on an attacker's fault model, which is an abstraction of the set of faults an attacker can perform (Barenghi, Breveglieri, Koren, and Naccache 2012). Using such a fault model is necessary to design both software and hardware countermeasures. Defining such an abstracted model requires a good understanding of the effects of the fault injection means. As many kinds of faults can be obtained even with a single fault injection technique, the practical efficiency of a countermeasure highly depends on the accuracy of the considered fault model. Thus, some experiments are necessary both to define realistic fault models and to guarantee the practical efficiency of a countermeasure.

In this paper, we experimentally evaluate the robustness of two software countermeasure schemes against fault injection on embedded programs. These two countermeasures have slightly different purposes and could be combined together. Both of them are designed at assembly code level and rely on providing some replacement sequences to strengthen some sensitive instructions. The first one, proposed in previous works (Moro, Heydemann, et al. 2014), aims at ensuring a fault tolerant execution. It covers almost all the instructions of the considered instruction set and has been formally proven resistant against an instruction skip fault model. The second one was proposed by Barenghi *et al.* (Barenghi, Breveglieri, Koren, Pelosi, et al. 2010). It uses an instruction duplication approach to perform a fault detection. It has been designed using a more generic fault model but covers a smaller set of instructions. The evaluation experiments that are conducted in this paper will enable us to determine some necessary conditions for an efficient implementation of these countermeasures and to highlight their possible limitations.

The rest of this paper is organized as follows. Section 2 provides an overview of some existing software countermeasure schemes and of the considered injection means for the experiments. Section 3 introduces the experimental platform and environment. Section 4 describes the two studied countermeasures and provides a practical evaluation of their robustness on simple assembly codes. Finally, Section 5 details some results obtained for the two countermeasures on some more complex codes from a FreeRTOS implementation.

2 Related works

This section reviews software countermeasures for embedded systems in 2.1 and motivates the use of an electromagnetic fault injection technique for the experiments in 2.2.

2.1 Software countermeasures

On embedded systems, software-only countermeasure bring some flexibility and avoid any modification on the underlying hardware. Against fault attacks, common countermeasure techniques directly come from software-implemented fault tolerance (SWIFT) techniques (Reis et al. 2005). Such countermeasure schemes include temporal redundancy, parity checking or checksum-based error detection (Barenghi, Breveglieri, Koren, and Naccache 2012). For cryptographic implementations, those SWIFT principles have mostly been applied at a function-level or algorithm-level (Oboril, Sagar, and Tahoori 2013). Otherwise, some algorithm-specific countermeasures (Joye 2012), some applicative countermeasures to protect Java Card applets (Sere, Iguchi-Cartigny, and Lanet 2011) or some combined software-hardware countermeasure schemes (Arora et al. 2005) have also been designed.

Those countermeasures are defined with respect to an attacker's model. Such a model provides a theoretical set of faults an attacker could produce. Since performing practical experiments on software countermeasures may require some advanced fault injection means and can be very time-consuming, fault models are also used to perform fault injection simulations (Theissing et al. 2013) or formal proofs (Moro, Heydemann, et al. 2014). Those simulations help to provide stronger guarantees about the efficiency of the tested countermeasures. However, certification processes include practical experiments ("Joint Interpretation Library Application of Attack Potential to Smartcards" 2009). Thus, the strongest guarantee can only be brought by performing practical experiments on real devices. To the best of our knowledge, no practical evaluation of the efficiency of some generic assembly-level countermeasures has been proposed yet.

2.2 Electromagnetic fault injection technique

Pulsed electromagnetic fault injection has been introduced in the last decade and has turned out to be an effective way to inject transient faults in a circuit's computation. Recent works, such as (Dehbaoui et

al. 2012) or (Moro, Dehbaoui, et al. 2013) tend to show that pulsed electromagnetic fault injection could enable to induce faults that are very similar to the faults obtained with clock glitches (Balasch, Gierlichs, and Verbauwhede 2011), voltage glitches (Zussa et al. 2013) or even laser shots on the logic part of a microcontroller (Trichina and Korkikyan 2010). Thus, we think that this electromagnetic fault injection technique should still be representative enough to get a good evaluation of the efficiency of the tested countermeasures.

3 Experimental setup

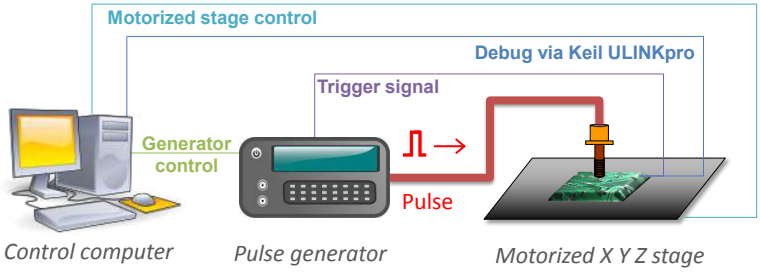

Figure 1: Electromagnetic fault injection bench

3.1 Pulsed electromagnetic fault injection

Conductors such as the rails of a power distribution network are one of the primary electromagnetic interferences risk factors for a circuit. They also act as antennas for the radiated electromagnetic pulse flux generated by a coil. This magnetic flux then induces an electromotive force in the power distribution network that leads to a violation of a circuit's timing constraints (Poucheret et al. 2011). In (Omarouayache et al. 2013), Omarouayache *et al.* studied magnetic probes built on the basis of small wire loops for the purpose of near-field injection. Their investigation have lead to some useful guidelines to design an electromagnetic antenna. They show that the antenna must be designed as wide-band components to transfer the electromagnetic power with the best efficiency. It was also shown that few loops must be used to optimize the field intensity, and the introduction of a point-sharpened ferrite in the middle of the loop concentrates efficiently the field for near-field operation. Such a magnetic probe enables to induce voltage drops in the target circuit. Those voltage drops then lead to violations of the timing constraints and to faults in the target circuit (Dehbaoui et al. 2012).

3.2 Target circuit

The chosen target is an up-to-date 32-bit microcontroller designed in a CMOS 130 nm technology. It is based on the ARM Cortex-M3 processor (Yiu 2009). Its operating frequency is set to 56 MHz without any cache memory. It is also important to mention that no prefetch buffer is activated. Thus, the full execution of some instruction can take several cycles. Cortex-M3 processors use a Harvard architecture and run the ARM Thumb-2 instruction set[1], which contains both 16-bit and 32-bit instructions. The target circuit embeds some basic security mechanisms against some low-cost fault injection techniques such as clock and voltage glitches. Some interrupt vectors can handle several hardware faults and can be used for a basic fault detection.

3.3 Electromagnetic fault injection bench

Figure 1 shows an architectural view of the electromagnetic fault injection platform. It is based on a high speed voltage pulse generator and uses a coil with few turns (diameter of 500 µm) as injection antenna.

[1]See ARM Architecture Reference Manual - Thumb-2 Supplement, 2005

The pulse generator is used to deliver voltage pulses (from −210 V to 210 V) to the magnetic coil. It has a constant rise and fall transition time of 2 ns. For our experiments, the pulses' width is set to 10 ns. The target circuit is mounted on a high-accuracy X Y Z motorized stage. The position of the injection antenna is the same for all the experiments of this paper, it has been found by a trial-and-reset approach. This bench also includes some standard control elements such as a PC, an oscilloscope and a Keil ULINKpro debug system. The computer sends pulse injection parameters to the pulse generator. It also controls the target board by using the Keil µVision UVSOCK library[2]. Since the microcontroller is restarted before injecting a fault, every fault injection attempt requires about 1 s. This fault injection bench and the influence of the different experimental parameters have been presented in more details in previous works (Moro, Dehbaoui, et al. 2013).

4 Experimental evaluation of the countermeasures

The following section first introduces the faults that can be obtained with our experimental setup in 4.1 and the approach we use for the evaluation in 4.2. Then, it provides an experimental evaluation of the two studied countermeasures on a very simple assembly code in 4.3 and 4.4.

4.1 Preliminaries about the fault model

The fault injection technique we use enables to induce violations of the timing constraints of an integrated circuit (Dehbaoui et al. 2012). For a microcontroller, bus transfers from the Flash memory are the operations that requires the longest time in a clock cycle (Moro, Dehbaoui, et al. 2013). Thus, the bus transfers from the Flash memory can easily be corrupted by using delay faults. The Flash memory contains both instructions and data. Thereby, two pipeline stages may be hit by such a technique: the *fetch* stage (for every instruction) and the *decode* stage. In the *fetch* stage, the circuit fetches 32 bits of data from the instruction memory at every clock cycle. Nevertheless, the Thumb-2 instruction set contains both 16-bit and 32-bit instructions. Thus, two instructions at a time might be corrupted. Moreover, *load* instructions also fetch a piece of data in the *decode* phase. Thus, this piece of data can be corrupted too.

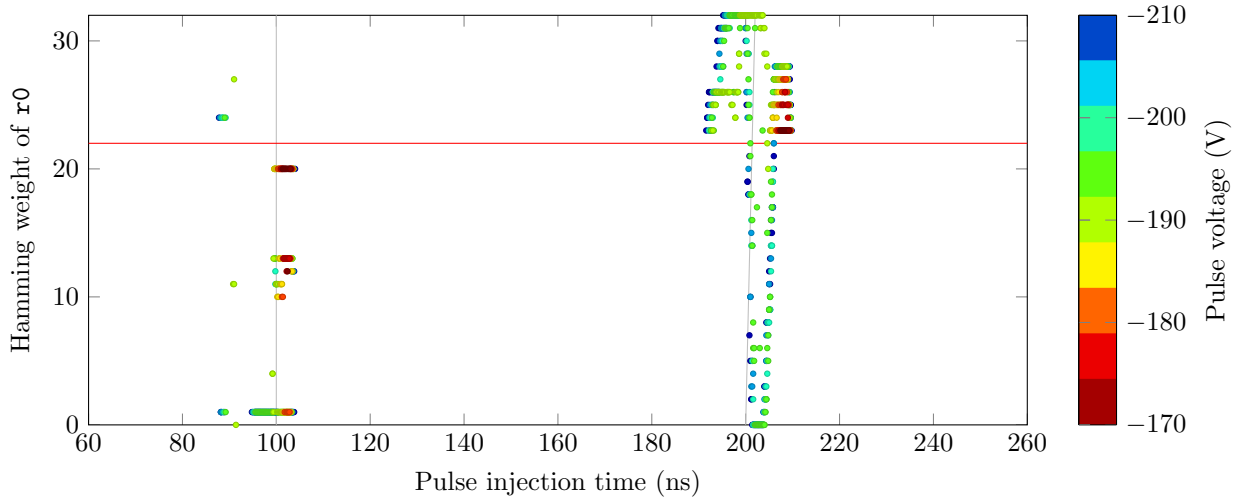

Figure 2: Fault injection results for a `ldr r0,=0xCAFECAFE` instruction

To highlight these two trends, we performed a fault injection experiment on a single 16-bit `ldr` instruction. In this example, the target instruction is a `ldr r0,[pc,#40]` PC-relative load instruction that has been generated by the `armasm` assembler from the macro `ldr r0,=0xCAFECAFE`. This loaded value has been chosen because it is a very specific value that cannot been found anywhere else in the memory or cannot be the

[2]Keil UVSOCK: http://www.keil.com/appnotes/docs/apnt_198.asp

output of almost all the possible instruction replacements. Moreover, some of the hardware exception handlers enable to access the address of the instruction that triggered the exception. We use this piece of information to define the time intervals that will be swept in our experiments. For this experiment, we performed a fault injection for 9 voltage values (from −210 V to −170 V) and over a 200 ns time interval. Another analysis we performed with positive voltages led to the same patterns for this target instruction. Such a 200 ns time interval is very long in comparison to the 17.8 ns clock period. This can be explained both by the fact that we need to cover the three pipeline stages of the instruction and by the fact that several clock cycles are necessary for the memory fetches. This time interval has been swept by steps of 200 ps. For every injection time, the fault injection process has been performed two times. The fault injection results are presented on Fig. 2. This graph shows the Hamming weight of the output values in `r0` when no exception has been triggered depending on the pulse voltage and the pulse injection time. The Hamming weight of the `0xCAFECAFE` expected value is 22. On this figure, we can clearly distinguish two groups of output faults, located around two injection times: those around around 100 ns and those around 200 ns. Their distribution of faulty values is very different. The first one corresponds to the *fetch* stage. Since the instruction *opcode* is corrupted, very few instruction corruptions lead to a valid new instruction. Most of them generate an invalid instruction that triggers an exception. Among the instruction *fetch* corruptions that led to a fault in `r0`, some instructions have also been transformed into branch instructions. The second group corresponds to the *decode* phase. Since the loaded word is corrupted, we can observe a much bigger diversity in the faulty outputs for this injection time. One has to note that such kind of results are not specific to electromagnetic glitches and have been obtained for other fault injection means (Balasch, Gierlichs, and Verbauwhede 2011; Trichina and Korkikyan 2010).

4.2 Evaluation approach

We need to define a relevant metric to evaluate the efficiency of the countermeasures. Since the countermeasure sequences add some instructions, the time to execute a full countermeasure sequence becomes longer than the time to execute the initial instruction. Thus, the number of vulnerable points, *i.e.* the injection times for which a fault injection attempt is successful, may also increase. Comparing the percentage of faulty outputs could appear to be a solution to compare two data sets with different numbers of measurements. Nevertheless, we assume that the most meaningful metric for such a comparison is the number of faults that have been obtained on the destination register. The countermeasure is really effective if it can overcome the fact that some new vulnerable points are added and if it can decrease this number of vulnerable points. Thus, comparing the number of faulty outputs is probably the most relevant metric for an attacker since it indicates the number of potential vulnerabilities on an embedded code.

Moreover, it is important to mention that we analyzed several metrics such as the global number of faults on any register or the number of faults that match an instruction skip for every experiment. It happened very frequently that several metrics show exactly the same pattern. For clarity purposes on the curves presented in this paper, we chose the metrics we assume to be the most significant for each experiment when the other relevant metrics showed the same trend.

4.3 Fault tolerance countermeasure

This countermeasure aims at providing a fault-tolerant replacement sequence for most of the instruction of an instruction set (Moro, Heydemann, et al. 2014). Such a countermeasure has been designed to be tolerant to any single instruction skip and does not provide any protection to the data flow. An example of the use of this countermeasure (from Moro, Heydemann, et al. 2014) is given in Listing 1. In this example, the replacement sequence mimics the effect of a `bl` instruction by putting the return pointer into the link register `lr` and branching to the destination function. Even if no fault is injected, the subroutine is only called once, since the return pointer is set after the two `b` instructions.

Listing 1 : Fault tolerance countermeasure for a `bl function` instruction

```
1     adr  r1, return_label
2     adr  r1, return_label
3     add  lr, r1, #1  ; Thumb mode requires the
4     add  lr, r1, #1  ; last bit of LR to be set
```

```
5       b       function
6       b       function
7   return_label
```

In Listing 1, the two `adr` and the two `b` instructions are encoded with their 16-bit encoding by default. In order to force the `armasm` assembler to use a 32-bit encoding, one can use the `.w` suffix after the instruction mnemonic (`adr.w`) or use the registers `r8` to `r14`. Indeed, since the two `add` instructions use the `lr` (`r14`) register, they are necessarily encoded with a 32-bit size.

Thus, we performed some fault injection experiments on four codes: a `bl` instruction without countermeasure (100 ns time interval by steps of 200 ps), a `bl.w` instruction with forced 32-bit encoding without countermeasure (100 ns), the replacement sequence from Listing 1 (400 ns) and this replacement sequence with forced 32-bit encoding (400 ns). Several values were used for the pulse voltage, from −210 V to −170 V and from 120 V to 150 V by steps of 5 V. The target circuit crashes for voltages over 150 V, and no faults were obtained for pulse voltages between −160 V and 120 V. In this experiment, the subroutine that is called only modifies `r0`. Thus, we analyze the number of faults in `r0` at the end of the experiment to evaluate the countermeasure. The *faults on any register* curves correspond to an output in which at least one register contains a value different from the expected one.

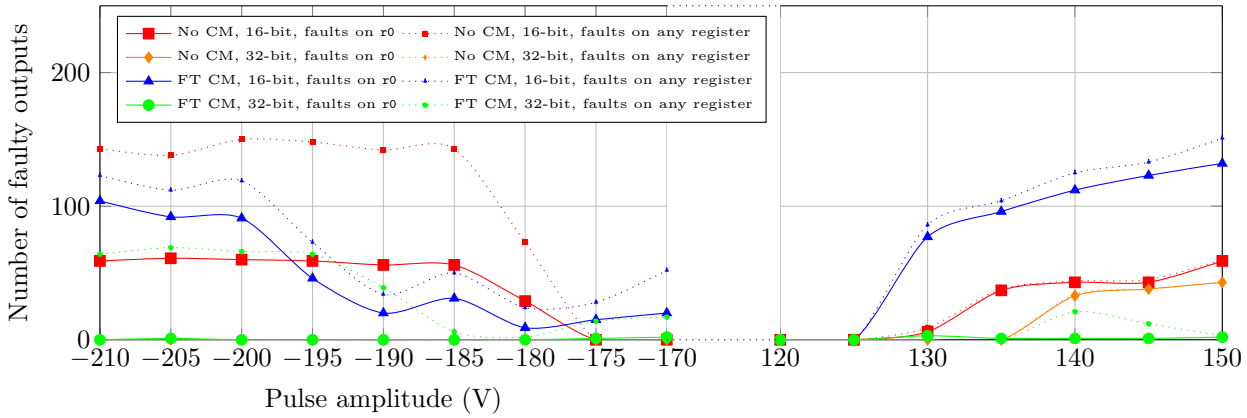

Figure 3: Fault injection results for the fault tolerance countermeasure

The fault injection results are shown on Fig. 3. First, we can observe that the curves related to the number of faults in `r0` and the number of faults on at least one register follow the same patterns, which tends to prove the relevance of choosing the number of faulty outputs in `r0` as a relevant metric. Then, we can also observe that applying the countermeasure without forced 32-bit encoding does not seem efficient. Indeed, this countermeasure has not been designed to be resistant to two consecutive instruction corruptions. Because of the memory alignment in the experiment, in the replacement sequence the first two `adr` instructions and later the last two `b` instructions are loaded in a single *fetch* stage. Thus, it seems that such double corruption happened. Moreover, we can also observe that no faulty output has been obtained for pulses with a negative voltage on a single 32-bit `bl.w` instruction. Nevertheless, such an instruction could still be faulted by using pulses with a positive voltage. An explanation for such result can be found in the way instructions are encoded. The 16-bit subset of the instruction set is very compact (most of the 16-bit values correspond to one instruction) while the 32-bit subset is very sparse: very few bit flips can change a 16-bit instruction into another instruction, but this assertion is not true for a 32-bit encoding. Finally, for the experiment with a fault tolerance countermeasure and a forced 32-bit encoding, some faults on other registers that are due to instruction *fetch* corruptions and very few faults on `r0` have still be obtained. To sum up, this countermeasure appears to be very effective for both positive and negative glitches. Applying the countermeasure scheme with a forced 32-bit encoding is a necessary condition to guarantee its efficiency.

4.4 Fault detection countermeasure

This countermeasure aims at detecting any single fault, including instruction skips, some cases of instruction replacements and faults on the data flow. It is based on duplicating the execution of an instruction and

storing its results in another extra register (Barenghi, Breveglieri, Koren, Pelosi, et al. 2010). Then, a comparison is done to detect any difference between the two destination registers. If an error is detected, the program branches to an error handler subroutine. This countermeasure can directly be applied to several ALU instructions. Nevertheless, it is not yet applicable to some more special instructions such as branch instructions or instructions that use the flags. As an example, the countermeasure for a `ldr` instruction (from Barenghi, Breveglieri, Koren, Pelosi, et al. 2010) is given in Listing 2. In this code example, a value is loaded from the Flash memory. The address of this value is relative to the program counter.

Listing 2 : Fault detection countermeasure for a `ldr` instruction

```
1 ldr     r0, [pc, #40] ; initial load instruction
2 ldr     r1, [pc, #38] ; duplicated load instruction
3 cmp     r0, r1        ; comparison between r0 and r1
4 bne     error         ; if r0 != r1, raise an error
```

We performed some fault injection experiments on four codes: a single `ldr` instruction that loads `0xCAFECAFE` (150 ns time interval by steps of 200 ps), a single `ldr.w` instruction with a forced 32-bit encoding (150 ns), the replacement sequence presented in Listing 2 (300 ns) and the same replacement sequence with a forced 32-bit encoding for every instruction (500 ns).

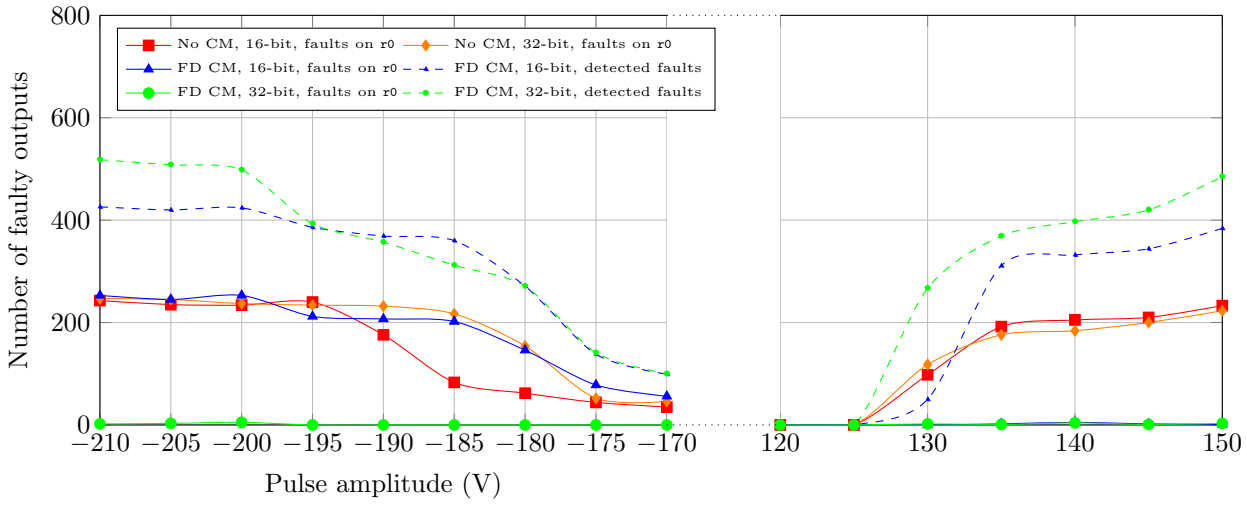

Figure 4: Fault injection results for the fault detection countermeasure

The fault injection results are presented in Fig. 4. The *detected faults* curves show the number of calls to the `error` subroutine. From a black box approach, we can observe that applying the countermeasure without forced 32-bit encoding creates more vulnerable injection times than the initial single `ldr` instruction and does not bring any security for negative glitches. Nevertheless, this countermeasure seems to work very well for positive glitches. Finally, the countermeasure scheme appears to be very effective with a 32-bit encoding of the instructions, it can handle both types of faults presented in 4.1, either on the data flow or the control flow.

5 Evaluation of the countermeasures on a FreeRTOS implementation

5.1 FreeRTOS and target implementation

This section gives details about some experimental results that have been obtained on a FreeRTOS-MPU implementation. FreeRTOS is a portable open-source real-time operating system (RTOS) for embedded devices. It it written in C and has been designed to be very simple to provide a convenient set of tools to design real-time applications. In the following experiments, the FreeRTOS-MPU implementation we use

has been built with the Keil MDK-ARM compiler. FreeRTOS is a multitasking operating system. It uses a scheduler to decide which task should be executing. At every interrupt from the system timer, this scheduler gives processing time to the task with the highest priority. FreeRTOS-MPU is a special port of FreeRTOS that uses the Memory Protection Unit (MPU) and the hardware privilege levels of the Cortex-M3 processor. It is able to create tasks in either privileged or unprivileged mode.

We chose to use a FreeRTOS implementation to show the practical interest such countermeasures could have for real-world complex projects that cannot be directly designed in assembly language. Indeed, these countermeasures could directly be applied to a compiled binary to reinforce an embedded system's resistance to fault attacks. A use case for the fault tolerance countermeasure is shown in 5.2, another one for the fault detection countermeasure is provided in 5.3.

5.2 Fault tolerance countermeasure

At system initialization, the tasks are created and the processor runs in privileged mode. Then, before starting the first task, the `prvRestoreContextOfFirstTask` function is called. This function uses several types of instructions and sets the execution context to run the first task. It also switches the processor to unprivileged mode if the first task is an unprivileged task. If the systems runs no privileged task, the processor never switches back to privileged mode since the scheduler also runs in unprivileged mode. In particular, this function is theoretically vulnerable to a fault attack: an instruction skip attack can skip the `msr`[3] instruction that switches to unprivileged mode. The most sensitive part of this function is shown on Listing 3.

Listing 3 : End of the `prvRestoreContextOfFirstTask` function

```
1 msr control, r3     ; switches to unprivileged mode
2 msr psp, r0         ; initializes the stack pointer
3 mov r0, #0
4 msr basepri, r0     ; base priority mask register
5 ldr lr, =0xfffffffd
6 bx  lr              ; returns to Thread mode
```

We performed some fault injection experiments on the whole function, either without countermeasure (2 µs time interval) or with the fault tolerance countermeasure and a forced 32-bit encoding applied to every instruction (5 µs). The results are presented on Fig. 5. They show a very mixed efficiency for the countermeasure on this code, with a good efficiency only for positive glitches. Such a result might be explained by the fact that this countermeasure has been designed for an instruction skip fault model. For some parts of the tested code, this fault model may be too simplified and may be an incorrect abstraction for the injected faults. Indeed, the instruction skip fault model has been observed for different experimental configurations on several targets (Dehbaoui et al. 2012; Barenghi, Breveglieri, Koren, and Naccache 2012). Nevertheless, some recent research papers have shown that instruction skips could be a specific case of replacements in the instruction binary code (Balasch, Gierlichs, and Verbauwhede 2011; Moro, Dehbaoui, et al. 2013). The instruction skip fault model can probably be a good abstraction on simple codes but visibly lacks of relevance for more complex codes. Thus, defining a more accurate fault model seems to be a prerequisite for any future improvement of this countermeasure.

5.3 Fault detection countermeasure

Every task has its own level of priority. A constant named `configMAX_PRIORITIES` is used to set the maximal level of priority that tasks can take. If the system tries to create a task with a higher level of priority, the task is created with the priority level contained in `configMAX_PRIORITIES`. Moreover, the `xTaskCreateRestricted` function takes as input a structure which contains the task parameters and uses the content of this structure to call the `xTaskGenericCreate` function. In such a structure, the `uxPriority` integer is used to define the task's priority. Since for the tasks that are created at the system's initialization those configuration structures are generally stored in the Flash memory, the `uxPriority` integer is generally loaded with a `ldr` instruction. Thus, a fault injection attempt may corrupt this `ldr` instruction and result into a priority elevation for a specific task. The target code is presented in Listing 4.

[3] `msr` moves the contents of a general-purpose register into a special register

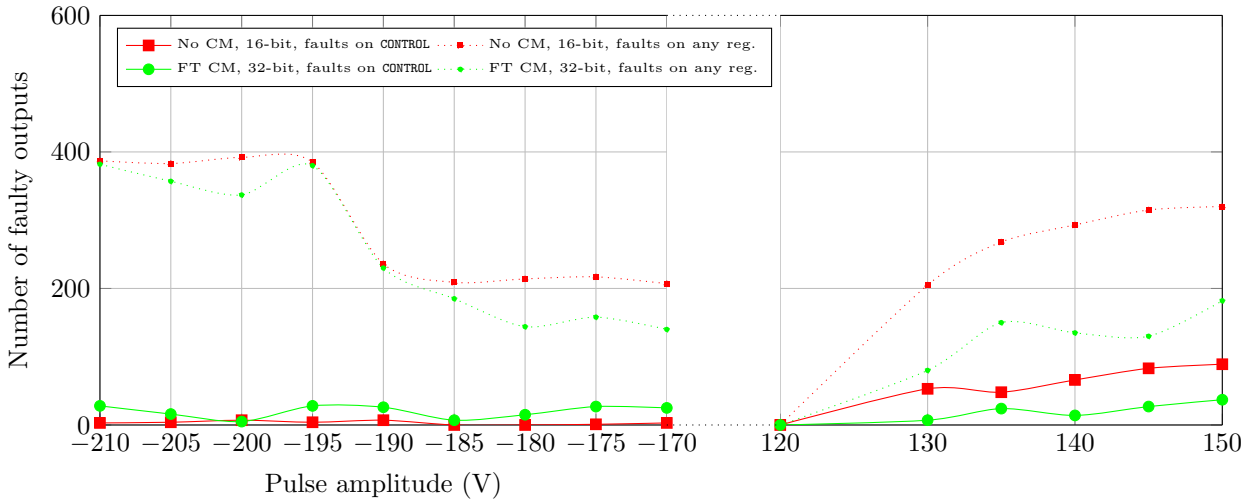

Figure 5: Fault injection results on the `prvRestoreContextOfFirstTask` function of a FreeRTOS implementation

Listing 4 : Last instructions before the call to `xTaskGenericCreate`

```
1 ldr  r0, [r0, #0]     ; loads uxPriority in r0
2 str  r0, [sp, #0]     ; puts uxPriority on the stack
3 movs r3, #0           ; null pointer (parameters)
4 movs r2, #128         ; stack depth for the task
5 movs r1, #0           ; empty string (task's name)
6 ldr  r0, =address_task_function
7 bl   xTaskGenericCreate
```

We performed some fault injection experiments on the 14 assembly instructions (which include 3 `ldr` instructions) that set the input arguments for the function, either without countermeasure (1 µs time interval) or with the fault detection countermeasure and a forced 32-bit encoding applied only on the `ldr` instructions (2 µs) or with the fault detection countermeasure and a forced 32-bit encoding on all the instructions (4 µs). The results are presented on Fig. 6. They show an average efficiency when only applied to `ldr` instructions. The remaining faulty outputs are due to the corruption of other unprotected instructions. Indeed, the countermeasure is very effective when applied to every instruction: less than 20 faulty outputs have been obtained for every tested voltage.

6 Conclusion

In this paper, we have provided a practical study of two assembly-level software countermeasure against fault injection attacks. Even if those countermeasures are theoretically secure, it turns out that the level of security they add could be nullified if their implementation on a target platform is not performed in the right way. On this target platform with a variable-size instruction set, we need to make sure that no more than one instruction at a time is loaded at every clock cycle. This evaluation has also been performed on more complex codes from a FreeRTOS implementation.

The fault tolerance countermeasure has been very effective to protect an isolated subroutine call instruction. Thus, it seems such a sensitive instruction can be significantly reinforced against fault attacks. Yet, on a complex code, its results have been very mixed. Since this countermeasure has been formally proven resistant to instruction skips, its main limitation appears to be due to its considered fault model, which is probably too simplistic and does not provide a good enough coverage of the induced faults. A deeper understanding of the faults that can be produced in practice is necessary to define a more accurate fault model and then improve this countermeasure.

The fault detection countermeasure has been designed to protect a smaller set of instructions. The

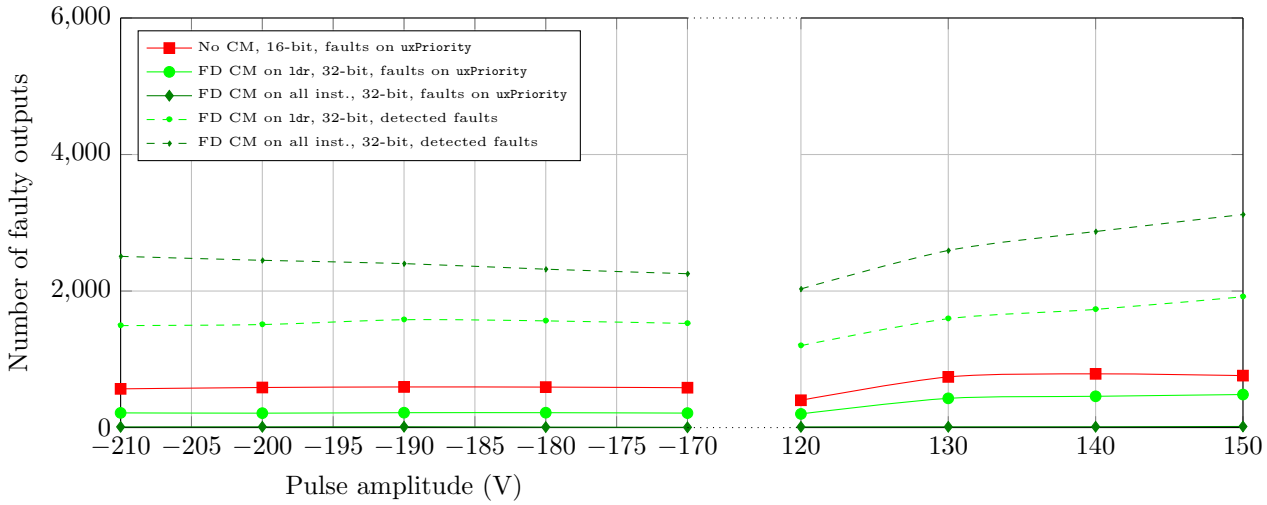

Figure 6: Fault injection results on the instructions before the call to the `xTaskGenericCreate` function of a FreeRTOS implementation

countermeasure has been very effective on the considered test cases. On a more complex code which only contains instructions for which a fault detection approach can be used, the countermeasure greatly increases the security level. However, its main drawback appears to be its limitation to a restricted set of instructions. Since the experimental results have shown a very good efficiency, this countermeasure needs to be extended to a larger set of instructions.

A possible extension of this work could evaluate the impact of such assembly-level countermeasures on the side-channel leakages of the circuit and study whether they can be combined with other software countermeasures against leakages.

References

Arora, D., Ravi, S., Raghunathan, A., and Jha, N.K. (2005). "Secure Embedded Processing through Hardware-Assisted Run-Time Monitoring". In: *Design, Automation and Test in Europe*. IEEE, pp. 178–183. DOI: 10.1109/DATE.2005.266 (cit. on p. 2).

Balasch, Josep, Gierlichs, Benedikt, and Verbauwhede, Ingrid (2011). "An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs". In: *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, pp. 105–114. DOI: 10.1109/FDTC.2011.9 (cit. on pp. 1, 3, 5, 8).

Barenghi, Alessandro, Breveglieri, Luca, Koren, Israel, and Naccache, David (2012). "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures". In: *Proceedings of the IEEE* 100.11, pp. 3056–3076. DOI: 10.1109/JPROC.2012.2188769 (cit. on pp. 1, 2, 8).

Barenghi, Alessandro, Breveglieri, Luca, Koren, Israel, Pelosi, Gerardo, and Regazzoni, Francesco (2010). "Countermeasures against fault attacks on software implemented AES". In: *Proceedings of the 5th Workshop on Embedded Systems Security - WESS '10*. New York, New York, USA: ACM Press, pp. 1–10. DOI: 10.1145/1873548.1873555 (cit. on pp. 2, 7).

Bhasin, Shivam, Selmane, Nidhal, Guilley, Sylvain, and Danger, Jean-Luc (2009). "Security evaluation of different AES implementations against practical setup time violation attacks in FPGAs". In: *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*. IEEE, pp. 15–21. DOI: 10.1109/HST.2009.5225057 (cit. on p. 1).

Boneh, Dan, DeMillo, Richard A, and Lipton, Richard J (1997). "On the Importance of Checking Cryptographic Protocols for Faults". In: *Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*. EUROCRYPT'97 1233, pp. 37–51 (cit. on p. 1).

Dehbaoui, Amine, Dutertre, Jean-Max, Robisson, Bruno, and Tria, Assia (2012). "Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES". In: *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, pp. 7–15. DOI: 10.1109/FDTC.2012.15 (cit. on pp. 1–4, 8).

"Joint Interpretation Library Application of Attack Potential to Smartcards" (2009). In: (cit. on p. 2).

Joye, Marc (2012). "A Method for Preventing "Skipping" Attacks". In: *2012 IEEE Symposium on Security and Privacy Workshops*. IEEE, pp. 12–15. DOI: 10.1109/SPW.2012.14 (cit. on p. 2).

Karaklajic, Dusko, Schmidt, Jorn-Marc, and Verbauwhede, Ingrid (2013). "Hardware Designer's Guide to Fault Attacks". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.12, pp. 2295–2306. DOI: 10.1109/TVLSI.2012.2231707 (cit. on p. 1).

Moro, Nicolas, Dehbaoui, Amine, Heydemann, Karine, Robisson, Bruno, and Encrenaz, Emmanuelle (2013). "Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller". In: *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, pp. 77–88. DOI: 10.1109/FDTC.2013.9 (cit. on pp. 3, 4, 8).

Moro, Nicolas, Heydemann, Karine, Encrenaz, Emmanuelle, and Robisson, Bruno (2014). "Formal verification of a software countermeasure against instruction skip attacks". In: *Journal of Cryptographic Engineering*. DOI: 10.1007/s13389-014-0077-7 (cit. on pp. 2, 5).

Oboril, Fabian, Sagar, Ilias, and Tahoori, Mehdi B. (2013). "A-SOFT-AES: Self-adaptive software-implemented fault-tolerance for AES". In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. IEEE, pp. 104–109. DOI: 10.1109/IOLTS.2013.6604059 (cit. on p. 2).

Omarouayache, R., Raoult, J., Jarrix, S., Chusseau, L., and Maurine, P. (2013). "Magnetic microprobe design for EM fault attack". In: *Electromagnetic Compatibility (EMC EUROPE), 2013 International Symposium on*, pp. 949–954 (cit. on p. 3).

Poucheret, F, Chusseau, L, Robisson, B, and Maurine, P (2011). "Local electromagnetic coupling with CMOS integrated circuits". In: *2011 8th Workshop on Electromagnetic Compatibility of Integrated Circuits* (cit. on pp. 1, 3).

Reis, GA, Chang, Jonathan, Vachharajani, Neil, Rangan, R., and August, D.I. (2005). "SWIFT: Software Implemented Fault Tolerance". In: *International Symposium on Code Generation and Optimization*. IEEE, pp. 243–254. DOI: 10.1109/CGO.2005.34 (cit. on p. 2).

Sere, Ahmadou Al Khary, Iguchi-Cartigny, Julien, and Lanet, Jean-Louis (2011). "Evaluation of Countermeasures Against Fault Attacks on Smart Cards". In: *International Journal of Security and Its Applications* 5.2, pp. 49–60 (cit. on p. 2).

Skorobogatov, Sergei (2009). "Local heating attacks on Flash memory devices". In: *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*. IEEE, pp. 1–6. DOI: 10.1109/HST.2009.5225028 (cit. on p. 1).

Theissing, Nikolaus, Merli, Dominik, Smola, Michael, Stumpf, Frederic, and Sigl, Georg (2013). "Comprehensive Analysis of Software Countermeasures against Fault Attacks". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*. New Jersey: IEEE Conference Publications, pp. 404–409. DOI: 10.7873/DATE.2013.092 (cit. on p. 2).

Trichina, Elena and Korkikyan, Roman (2010). "Multi Fault Laser Attacks on Protected CRT-RSA". In: *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, pp. 75–86. DOI: 10.1109/FDTC.2010.14 (cit. on pp. 1, 3, 5).

Yiu, Joseph (2009). *The Definitive Guide To The ARM Cortex-M3*, p. 479 (cit. on p. 3).

Zussa, Loic, Dutertre, Jean-Max, Clediere, Jessy, and Tria, Assia (2013). "Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism". In: *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. IEEE, pp. 110–115. DOI: 10.1109/IOLTS.2013.6604060 (cit. on pp. 1, 3).